
# Abysssec Research

## 1) Advisory information

| | |
|---|---|
| **Title** | **: Adobe Shockwave Director tSAC Chunk memory corruption** |
| **Version** | **: dirapi.dll 11.5.7** |
| **Analysis** | **: http://www.abysssec.com** |
| **Vendor** | **: http://www.adobe.com** |
| **Impact** | **: Med/High** |
| **Contact** | **: shahin [at] abysssec.com , info [at] abysssec.com** |
| **Twitter** | **: @abysssec** |

## 2) Vulnerable version

**Adobe Shockwave Player version 11.5.7.609 and prior**

## 3) Vulnerability information

Class

**1- Memory Corruption**

Impact

**Successfully exploiting this issue allows remote attackers to cause denial-of-service conditions.**

Remotely Exploitable

**Yes**

Locally Exploitable

**Yes**

# 4) Vulnerabilities detail

## 1- Division by Zero:

Shockwave director file format is a kind of undocumented format based on riff format. In riff format every chunks start with a 4bytes identifier that specify ID of the chunk. For example pami, pamm, tASC are some of these identifiers in director files. After these 4bytes identifier 4bytes represent size of the chunk and next bytes are data with the length of the mentioned size.

Here is a simple sample chunk:

```
4C 46 44 4D 06 00 00 00 00 00 00 02 3A 7E
```

4C 46 44 4D is the identifier which is equal to MDFL in reverse order and 06 00 00 00 is size of the chunk equal to 6 bytes and then data of the MDFL chunk with 6bytes size.

There are some vulnerabilities exist in parsing of tSAC chunk in some unknown records. Our intended vulnerable function which is responsible in parsing of tSAC chunk is sub_68082AC0.

Here is the beginning of the function :

```
.text:68082AC0          sub     esp, 70h
.text:68082AC3          push    ebx
.text:68082AC4          push    ebp
.text:68082AC5          mov     ebp, [esp+78h+arg_0]
.text:68082AC9          push    esi
.text:68082ACA          push    edi
.text:68082ACB          push    ebp
.text:68082ACC          mov     edi, eax
.text:68082ACE          mov     ebx, ecx
.text:68082AD0          call    IML32_1414
.text:68082AD5          mov     esi, eax
.text:68082AD7          cmp     esi, 20h
.text:68082ADA          jg      loc_68082C84
.text:68082AE0          push    esi
.text:68082AE1          lea     eax, [esp+84h+var_24]
.text:68082AE5          push    eax
.text:68082AE6          push    ebp
.text:68082AE7          call    IML32_1409
.text:68082AEC          test    eax, eax
.text:68082AEE          jz      loc_68082C84
.text:68082AF4          mov     edx, [ebx+20h]
.text:68082AF7          lea     ebp, [edi+5Ch]
.text:68082AFA          mov     [esp+esi+80h+var_24], 0        ← crash
```

The above code reads some value directly from data of the tSAC chunk from some undocumented record by calling the IML32_1414 function and IML32_1414 function return the value to esi register. Later this value will be used as an index in the mov [esp+esi+80h+var_24], 0 instruction. The instruction simply initializes byte of the stack to zero.

The flaw exist in checking of esi register after calling IML32_1414. The code checks esi register if greater than 20h with 'cmp esi, 20h', 'jg loc_68082C84' instructions and make a conditional jump past mentioned instructions. But jg is a signed integer conditional jump and it means if esi that is under our control contains negative values the check does not perform correctly and it reaches to the crash point. By using ja that is a an unsigned version of jg the problem can be patched.

To reach this point we used a sample director file because of undocumented file format and perform some tests in CASt chunks. Here is a part of the chunk in hex :

```
74 53 41 43 1D 02 00 00 00 00 00 0F 00 00 00 AE 00 00 01 63 00 00 00 14 00 00 00 00 00 00 00 00 00
00 00 04 00 00 00 00 00 14 00 00 01 00 FF FF 11 11 00 00 00 00 00 00 00 0B 00 00 00 0B 00 00 00 0B
00 00 ...
```

74 53 41 43 is equal to CASt chunk with size of 21D and in some part of the chunk by using 01 00 FF FF 11 11 we intend the execution to reach the crash point and FF FF 11 11 will be stored to esi register as mentioned before.

To trigger this crash and other same flaws that happen a lot in this undocumented format we can simply implement a fuzzer to generate random tests. Here is our simple implementation that triggered many of them and we explained one of them as a sample.

```
#Abysssec.com
import sys

dirFile = open("c:\\sample.DIR", 'rb') # director file  resource

length = len(dirFile.read(-1))
dirFile.seek(0)

x = 0
while x < length + 4:
        fuzzFile = open("c:\\fuzzfile2\\test%s.DIR"%x,'wb')
        dirFile.seek(0)
        fuzzFile.write(dirFile.read(-1))
        fuzzFile.seek(x)
        fuzzFile.write("\xff\xff\xff\xff") # it can be0xFFFFFFFF or some other negative or random
        fuzzFile.close()
        x = x + 4

dirFile.close()
```